AUTOMATIC CONTROLLER SYNTHESIZER FOR DIFFERENT ROBOTIC PLATFORMS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Chuanwei Wu

August 2019

ABSTRACT

This thesis presents a software tool that transfers a user's high-level task into robot behaviors. The work presented in this thesis is implemented based on the Robot Operating system (ROS). It is an integration work built upon Linear Temporal Logic MissiOn Planning (LTLMoP) [4], Linear Temporal Logic stack (LTLstack) [13] and SmalL bUt Complete GROne Synthesizer (slugs) [3]. The upgrades are mainly focused on better user experience, implementing the possible behaviors on the different physical robots and taking the robots to work outside of the lab. The work can automatically provide feedback to the user. The feedback includes the possible failures in mapping between the automaton and related ROS nodes, the possible bugs in the ROS node itself and the possible inappropriate ROS initialization setup.

BIOGRAPHICAL SKETCH

The author was graduated from Dalian University of Technology in July 2009 with Bachelor of Art degree in English. After the completion of his training, Chuanwei worked in industry for five years. From July 2009 to June 2010, he was employed by Tianjin Alstom hydro co. ltd in the capacity of workshop manager assistant, from July 2010 to June 2014, he was employed by China Energy Recovery co. ltd in the capacity of on-site assistant project manager. After years working with engineers, Chuanwei decided that he wanted to be an engineer. He quit his job and moved to United state for further study in engineering.

The author was graduated from University of Colorado at Boulder in May 2017 with Bachelor of Science degree in Mechanical Engineering. After the author got his BS degree, he joined Verifiable Robotics Research Group in Cornell University under the supervision of Professor Hadas Kress-Gazit. His main research interest is focused on how to make the automatic controller synthesis process more user-friendly.

This thesis is dedicated to my wife, Aidi, who is always on my side. You have always been there for me and I couldn't have asked for more loving wife. And thank you for giving me the best gifts on earth, my daughter Bonnasune and my son Courasun.

ACKNOWLEDGEMENTS

Thanks to my advisor, Professor Hadas Kress-Gazit, for her patience and mentoring along the way. Without her help, none of this is possible.

Aside of my dedication, thanks to my wife, Aidi, for proofreading the thesis for me.

Thanks to my parents and my parents-in-law, who gave me every opportunity to become whatever I wanted. Thank you for your selfless support and believing in me.

Thanks to the members of my research group and lab mates along the way: Ji Chen, Kai Weng (Catherine) Wong, Jijie (Gigi) Zhou, Adam Pacheck, Scott Hamill and Thais Campos for great working environment and positive encouragement.

And most importantly, praise be to God, who is my light and my salvation. May this work bring glory to you, forever and ever.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Nowadays, there is a gap between the demand of using a robotic platform and the ability of doing so. People are trying to bridge the gap by making the robotic platform more and more user-friendly. The programming of the robotic platform can be hard, and most of the time the robot can only be operated in a lab environment with additional sensors attached to it. This thesis tries to make the robot easier for the user by lowering the programming requirements. And also the tool presented in this thesis tries to operate the robots outside of the lab. The tool is integrated from Linear Temporal Logic MissiOn Planning (LTLMoP) [4],Linear Temporal Logic stack (LTLstack) [13] and SmalL bUt Complete GROne Synthesizer (slugs) [3].

The programming of the robot could be easier if we could code in high-level tasks instead of low-level controllers. The automaton allows the user to reuse the existing low-level controller. Users can make the automaton based on the high-level specifications in linear temporal logic using existing toolkit, such as slugs [3]. In this way, the robot would be able to know what to do and where to go by itself. LTLMoP [4] can reduce the workload by allowing the user to input Structured English and a map, in which the robot is operated in, to make the automaton. The map is turned into high-level specifications automatically by LTLMoP.

From automaton to the robotic platform behaviors, we used LTLstack to map the outputs of the controller to specific programs which control the robot [13]. With the latest version of LTLstack presented in this thesis, users don't need to map the outputs to the low-level controller manually anymore. LTLstack can

make the connections and the robots are able to perform the desired behavior accordingly. LTLstack is built based on robot operating system (ROS) [10], however the robot does not need to have ROS on board. Any connection protocol can be used to send message to the robot.

Contribution:

In this work, the author utilized the structure and libraries of ROS to establish a connection between Structured English and physical world robot behaviors. Specifically, the author: (1) Added *.slugsin* to the outputs of LTLMoP besides *.ltl* and *.smv*. With *.slugsin*, LTLstack is able to directly use this output for the further steps. (2) Cancelled all the processes after *.slugsin* is generated in LTLMoP. The cancelled processes are realized by LTLstack, since LTLstack can be used on more generalized robotic platforms. (3) Limited the queue size for ROS publisher nodes in LTLstack. When the physical robot is operated, there is communication delay between the roscore and the robot. Limiting the queue size helps to reduce the delay significantly. (4) Updated the LTLstack to ROS kinetic version. So the tool can be used on newer and more advanced robots with Ubuntu 16.04 built in. (5) Added an additional TCP protocol package. In this way, the tool can be used on the robots without ROS installed.

CHAPTER 2

BACKGROUND

The work presented in this thesis is an integration of numbers of existing tools, GR(1) Synthesis for LTL Specification, Structured English, temporal Logic Robot Mission Planning for Slow and Fast Actions, Robot Operating System (ROS) and Adaptive Monte Carlo Localization (AMCL). With these theories and tools working together, the automatic controller synthesizer is created and used on different robotic platforms.

2.1 GR(1) Synthesis for LTL Specification

Linear temporal logic (LTL) is a modal logic with operators referring to time. For example, one can encode that a condition is always true, a condition will eventually be true, a condition will be true until another fact becomes true and so on [6].

LTL is built from a finite set of Boolean variables, AP, the logical operators ! and $\vee$, and the temporal modal operations X and $\mathcal{U}$ [9].

$$\varphi ::= \pi \mid !\varphi \mid \varphi \vee \varphi \mid X\varphi \mid \varphi \mathcal{U} \varphi \tag{2.1}$$

where $\pi \in AP$, ! is negation, $\vee$ is disjunction, X is the ”next” and $\mathcal{U}$ is ”until”. Implication ($\rightarrow$), equivalence ($\leftrightarrow$), ”eventually” (F), ”always” (G) are also used. An LTL formula is evaluated over an infinite sequence of truth evaluations of variables in AP.

Reactive synthesis is an automated process to obtain a reactive system from its temporal logic specification. Piterman et al. [8] have suggested the General Reactivity of Rank 1 (GR(1)) fragment of LTL, which helped to short the synthesis time. Of the many synthesis approaches available, GR(1) has found widespread use for applications in robotics and control. Reasons for this success include its comparatively low complexity, and its amenability to symbolic computation using binary decision diagrams (BDDs) [3]. GR(1) is a strict assume/guarantee subset of LTL, comprised of constraints for initial states φ_i, safety formulas φ_t over the current and successor state, and liveness constraints φ_g. Intuitively, if the assumption φ^e is satisfied by the environment, the guarantee φ^s is satisfied by the system.

$$\begin{aligned} \varphi^e &= \varphi_i^e \wedge \varphi_t^e \wedge \varphi_g^e \\ \varphi^s &= \varphi_i^s \wedge \varphi_t^s \wedge \varphi_g^s \\ \varphi &= \varphi^e \rightarrow \varphi^s. \end{aligned} \tag{2.2}$$

In this thesis, the automaton is built based upon the combination of user's inputted specifications and the workspace topology. The tasks can be inputted to the software in either Structured English [7] or Linear Temporal Logic (LTL) [9], while the topology is provided to the software based on the static map in which the robot is operating in. SmalL bUt Complete GROne Synthesizer (slugs) [3] is used to synthesize the automaton. The environmental assumptions are the constrains which cannot be controlled, such as whether the light is on and in which state the human is at the next time step. The system guarantees are the constrains which are limited by the ability of the robots, such as a ground robot can only move on a 2D map and how many different sounds a robot can make at the same time.

2.2 Structured English

Structured English is the use of the English language with the syntax of structured programming. It is used to communicate with the design of a computer program to the user with no programming experience. It is built from English words and can be broken down into logical steps. Structured English gets benefits of both programming logic and natural language: program logic is able to keep the documentation precise, while the nature language is easier for the readers to understand. There are several types of sentences that can be written in this grammar: initial conditions, environment assumptions, motion/action requirements and conditional sentences. Here is the syntax of commonly used Structured English [7],

1. Valid initial condition sentences are:

   "Environment can start with ϕevn", in which φ_{evn} is the initial condition for the sensor propositions.Similarly, "Robot can start with ϕ_{region} and ϕ_{action}".

2. Valid environment assumptions and motion/action requirements are:

   "Always φ", same meaning as Gφ in LTL;

   "Go to φ", same meaning as GFφ in LTL;

3. There are three types of conditional sentences that are allowed in Structured English:

   "if condition then requirement", same meaning as $\phi_c \rightarrow \phi_r$ in LTL;

   "requirement unless condition", same meaning as $!\phi_c \rightarrow \phi_r$ in LTL;

   "requirement if and only if condition", same meaning as $\phi_c \leftrightarrow \phi_r$ in LTL.

2.3 Temporal Logic Robot Mission Planning for Slow and Fast Actions

In the process of a continuous execution of the controllers by a physical robot, several low-level controllers are triggered at the same time. If these low-level robotic behaviors have different time durations, the system might pass through potentially unsafe intermediate states. The reason is that the automaton assumes the robot has finished the task while the robot is still trying to finish. The execution has to take into account these durations to make sure the physical robot works properly [11].

The work from Raman et. al. [12] presents an algorithm that generates a hybrid controller such that the robotic behavior with different time lengths is safe. This algorithm is applicable for the case where robot actions are either fast or slow, denoting t_F and t_S respectively, with $t_F < t_S$. This method assumes that motion is the only slow controller. For the long time consuming robotic platform behavior, one more proposition ended with "ac" or "rc" is added to the specifications. Here, "rc" stands for region complete, meaning the desired sub-workspace is reached, while "ac" stands for action complete, meaning the desired action is completed. The feature added to the system is called action/-motion complete. It is a feature providing additional information to the system whether the current action/motion is finished or not [12]. In this way, the state is only changed when the current action/motion is finished. It is to avoid the situation that the automaton and robot are in different states.

2.4 Robot Operating System (ROS)

ROS is an open-source, meta-operating system for robots. It provides the services of an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. ROS is a peer-to-peer network of processes that are loosely coupled using the ROS communication infrastructure [10]. To use ROS, users created programs called nodes, the nodes are composed based upon custom or existing ROS packages. Each ROS node can send robot commands, update environment sensor information and transfer data. All the ROS nodes must connect to the ROS master called roscore. Through the ROS master, roscore, the message can be transferred between nodes via three different paths.

1. Topic

   To transfer message using topics, the node has to be either a subscriber or a publisher. The publisher provides message to the topic while the subscriber receives the message from the topic. Each topic can only transfer one kind of message at the same time, and the message type should be specified when the topic is initialized.

2. Service

   To transfer messages using a service, the node has to be either a service requester or a service provider. The service requester node sends the request to the provider and waits for the reply. For example, in the ROS package move_base [2], the service requester node can send request to the service provider node asking for the next valid waypoint. The waypoints

are calculated based on where the robot is and where the robot's goal is.

3. Action

   Action is a longer duration version of service. The nodes interact logic between action server and action requester is the same as service requester and the service provider. The difference is that action nodes transfer message via a special kind of topic, called action_topic.

2.5 Adaptive Monte Carlo Localization (AMCL)

Monte Carlo Localization is an algorithm for robots to localize using particle filter. Given a map, the algorithm estimates the position and orientation of the robot while the robot moves in the map. The algorithm uses the environment information gathered by the sensors on the robot to compare with the map. This result is used to determine the highest probability location of the robot in the map. Whenever the robot moves in the map, new information is gathered to update the possibility of where the robot is. Adaptive Monte Carlo Localization (AMCL) is a ROS package implemented with the Monte Carlo Localization theory [5]. In this thesis, AMCL is done by the ROS package ROS_AMCL [1]. The location of robot in this work is either provided via VICON motion capture system or AMCL.

CHAPTER 3

RELATED WORK

The tool presented in this thesis is an integration tool of several existing tools. The LTLMoP [4], LTLstack [13] and slugs [3] all have big impacts on this work.

3.1 LTLMoP

The LTLMoP [4] is used as the user interface of the tool, shown as Fig. 3.1

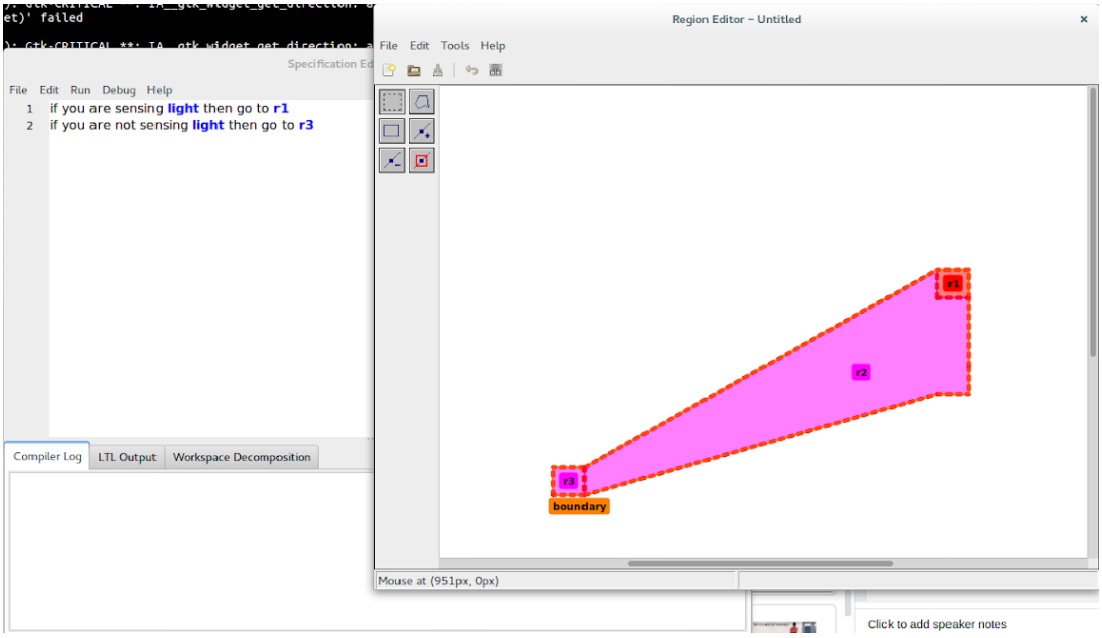

Figure 3.1: An example of the user interface used in LTLMoP

Users are able to input the map and tasks in Structured English into the LTLMoP. These inputs can be automatically transferred to high-level specifications in the form of LTL, in this way, the amount of workload is shrunken for the users. The LTLMoP generates *.smv* and *.ltl* as outputs. The automaton is generated based on the *.smv* and *.ltl*. After the automaton is generated from LTLMoP, LTLMoP

is able to simulate how the robot works in the map according to the high-level specifications.

The tool introduced in this thesis mainly changes two functions compare with the original version of LTLMoP. The tool is able to generate *.slugsin* automatically as additional output. In this way, LTLstack is able to directly use this output for the further steps. Another change, the author applied to the LTLMoP is that the author cancelled all the processes after *.slugsin*. The cancelled processes are realized by LTLstack, since LTLstack can be used on more generalized robotic platforms. After these two changes are applied, the tool is able to take Structured English/LTL specifications as input and generate *.slugsin* as output. As the finite state machine synthesis task is removed from LTLMoP, the runtime of LTLMoP is shorten.

3.2 LTLstack

LTLstack [13] is built based on ROS, it takes high-level specifications in the form of *.slugsin* as inputs to build automaton and outputs the robotic platform behaviors.

LTLstack utilizes slugs to build the automaton and find the next state with the interactive synthesis option. This synthesis option is a plugin that lets slugs execute a controller in an interactive way, such that it can be used as a tool for simulating the controller called from other tools [3].

From an automaton to the robotic platform behaviors, we have to find a path between the outputs of the controller and the specific programs which control

the robotic platform. LTLstack is able to build the path by triggering *executor* built based on ROS. LTLstack is also able to detect possible failures related to the mapping between the controller and the ROS nodes connected to the controller. The feedback information are provided to the user for the further programming modification possibilities [13].

The tool introduced in this thesis mainly changes the following three functions compare with the original version of LTLstack. In LTLstack, there is a limited queue size for ROS publisher nodes, but no limited queue size for ROS subscriber nodes. While simulation is applied using LTLstack, this setup works. Because all the ROS nodes and the roscore work on the same computer, the communication delay is negligible. When the physical robots are operated, the communication delay between ROS nodes and the roscore can be tens of seconds, even minutes. To fix this problem, We limited the queue size in the ROS nodes for not only publisher but also subscriber. In this way, the robot always run the latest command, so the delay can no longer be observed in the behavior.

The original version of LTLstack is composed based on ROS Indigo version, which works while applied with KUKA youbot. The author wants to apply LTLstack to newer and more advanced robots, for example, Jackal, turtlebot and so on. The newer and more advanced robot often comes with newer and more advanced build-in operating system, which cannot work with ROS Indigo. To fix this problem, the author upgraded the LTLstack to make it workable with ROS kinetic, the next version of ROS.

The other new feature added to the LTLstack is that it can be applied to the robot which is not using ROS. ROS is a worldwide popular operation system choice, however it is attached closely to Ubuntu Linux. Without Ubuntu Linux,

ROS cannot be installed on the robot. Some of the robots don't have enough capability to carry a Ubuntu Linux system on their mother boards. For this reason, they cannot work with the original version of LTLstack. The robot without ROS just means that they are not able to receive ROS message, it doesn't mean that they cannot receive message. The author added the TCP protocol package to the publisher ROS nodes in LTLstack, so these nodes are able to send commands to the robot without ROS. In this way, LTLstack is able to be applied to more robots.

3.3 Temporal Logic Planning toolbox (TuLiP)

Temporal Logic Planning toolbox (TuLiP) is a Python package for automatic synthesis of correct-by-construction embedded control software [14]. There are two key features of TuLiP: (1) TuLiP models the embedded control software synthesis problem as a game between the robot and the environment. TuLip handles the specification assented to be in the form of GR(1). (2) TuLiP used the method called "Receding Horizon Framework" to reduce the complexity of the planner synthesis. The idea is to compute the plan over a "shorter" horizon, beginning at the current state, implement the initial portion of the plan, and recompute the plan. This approach helps to reduce the big problem into several smaller problems, so the runtime is shrunken.

The tool presented in this thesis is using a different controller synthesizer to generate the automaton, slugs. Slugs is a generalized reactivity (1) synthesis tool pre-equipped with numbers of plugins that improve the quality of the solutions in many ways, such as faster response, lower computational cost and

error-resilience [3]. And the tool introduced has an additional function, which is mapping the existing automaton to the robot behaviors.

CHAPTER 4

APPROACH

The work-flow of the tool introduced in this thesis is shown as below in Fig. 4.1

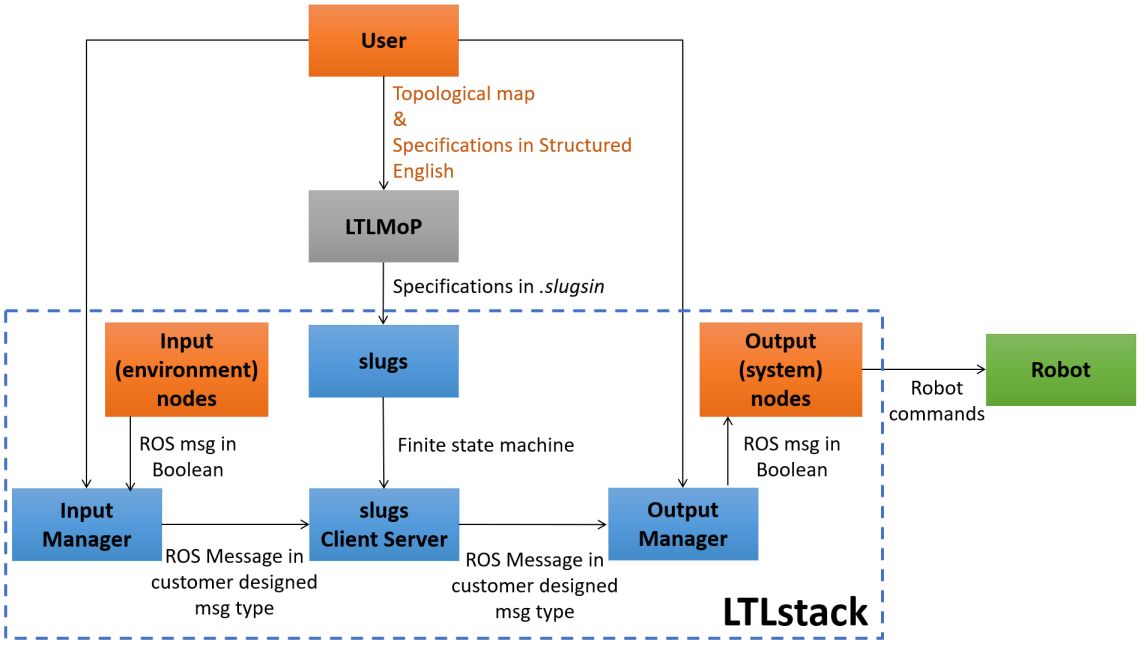

Figure 4.1: A diagram showing how the tool works

The orange parts in the diagram are the required inputs from the user. The blocks in the blue dash are the internal parts of LTLstack. The arrows indicates how the information is transferred, while the words next to them explain what information is transferred. More details are in Appendix A.

The specifications inputted by the user in Structured English are parsed into LTL specifications as described in Chapter 2. The map, where the robot is operating in, is turned into the related LTL specifications using `hsubConfigObjects.py` in LTLMoP. It takes the map and cuts it into several polygonal sub-workspaces first, and makes LTL specifications based on how

they are connected to each other [4]. The environment changes provided by the user are also transferred into LTL [4]. If the fast/slow option is turned on in LTLMoP, one more proposition ended with ac or rc is added to the specifications automatically to avoid potentially unsafe inter-mediate states by LTLMoP [11]. After LTLMoP generates the LTL specifications in *.slugsin*, LTLstack is triggered to synthesize the finite state machine.

Once LTLstack gets the high-level specifications in *.slugsin*, LTLstack gets the initial state information from the high-level specifications. With the given LTL specifications, the synthesis problem may able to be solved by an automaton if the automaton satisfies all the specifications [4]. In other words, if such automaton can be created, the specifications are realizable. Then LTLstack uses slugs [3] to generate the finite state machine.

In LTLstack, we created a ROS client server called `slugs_startup_server_base.py`. This file utilizes the interactive behavior built in slugs to create the automaton and gets the next states continuously. The current state of the robot and environment information is inputted into slugs, including where the robot is and all the relative environment sensors information. Using all these pieces of information together, slugs is able to calculate what is the next state, and the ROS client server is able to gain the information and save it for the next step. This process is triggered whenever the automaton is made and continues until the process ends.

LTLstack offers the possibility for the nodes to be organized, while the individual node has to be manually programmed [13]. The robot nodes are divide into two categories, the inputs and outputs. The environment (sensor) ROS nodes are related to the sensors information, such as where the robot is, whether

the environmental light is on or whether the demanded apriltag is observed by the robot. The environment (sensor) ROS node is programmed as subscriber. The system (actuator) ROS nodes are related to the robot behaviors. They send commands to the robots, such as what linear velocity should be applied, what angular velocity should be applied or whether the light on the robot should be turned on. The system (actuator) ROS node is programmed as publisher.

Once the automaton is made and all the user made ROS nodes are done, LTLstack turns `executor.py` on. This file triggers all the related files in the ROS structure, including `input_manager.py`, `output_manager.py` and `proposition_monitor.py`. The following diagram, Fig.4.2, explains how the LTLstack works after the automaton is made internally in detail,

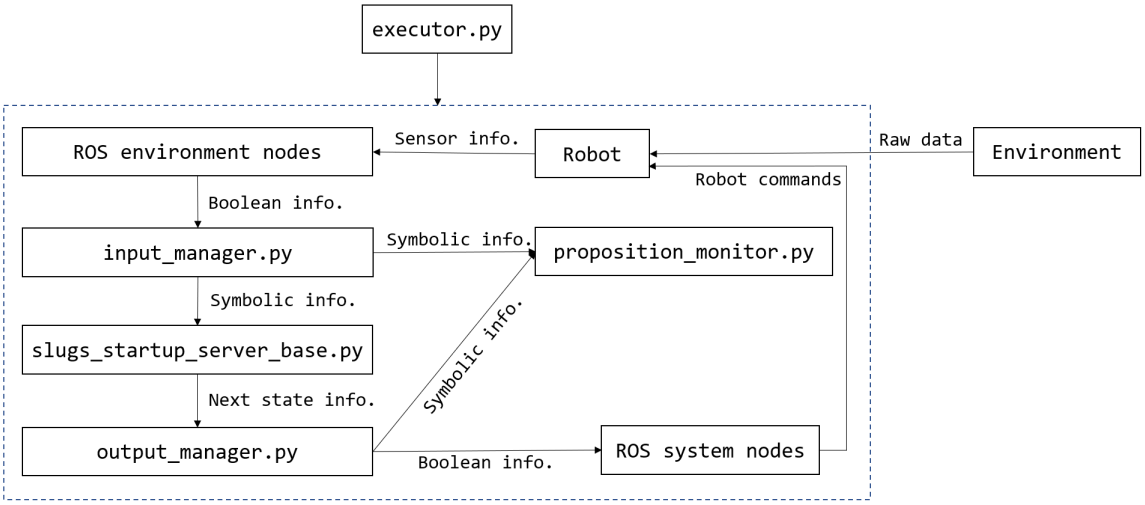

Figure 4.2: A diagram showing how LTLstack works after the automaton is made

The sensors information is subscribed and transferred to the related Boolean message by the environment (sensor) ROS nodes. `input_manager.py` is then

called to transfer the gathered environment message from ROS environment nodes into a new self-defined message type. The new message type contains two parts, the proposition names and their statuses. This message type helps the automaton state to track the situation of the robot as well as the environment sensors readings. `input_manager.py` sends the message to slugs ROS server to calculate the next state. `output_manager.py` requests slugs ROS server to provide next state information. This information is also stored as new self-defined message, containing the proposition names and their statuses. After `output_manager.py` heard next state message replied from the slugs ROS server, the `output_manager.py` dismantles the received next state information into several pairs. In each pair, there are a proposition name and a related Boolean status, true or false. Trough the topics, the commands coming from `output_manager.py` in the form of ROS message are published to the related ROS system nodes. The ROS system nodes then send robot commands to the robot. Then the robot can make robot behaviors according to the commands received. In the system nodes, there are two different kinds of behaviors, motions and actions. The motion ROS nodes are able to move the robotic platform from one place to another. These nodes are programmed using *move_base* [2], a ROS package, which is able to calculate how to get to the goal position based upon the given location of the robot and the desired goal. The limitation of *move_base* is that the algorithm cannot avoid entering a certain region. As shown in Fig.4.3, the package is able to control the robot to reach region r_1, r_2, r_3 or r_4.

But it cannot control the robot to reach r_1 from r_3, while avoiding r_2. This is due to the nature of *move_base*, it makes the way point for the robot to reach using local and global motion planner. Neither of these two planners can schedule the route while having a certain region blocked. To perform the robot behavior

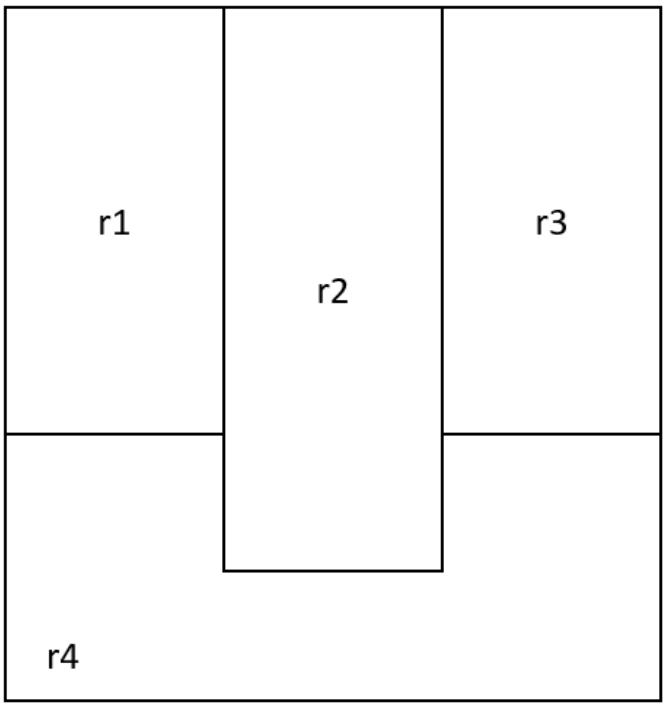

Figure 4.3: An example showing the limitation of *move_base*

of "reaching r_1 from r_3, while avoiding r_2", the user should define the robot dynamics specifically within the ROS system nodes instead of using *move_base*.

The action ROS nodes can vary from each other, including performing a pre-designed dynamics, making a sound from the speaker, and changing the color of the light on the robot.

`proposition_monitor.py` is made to visualize the propositions and their related Boolean status. This file displays the current state of the automaton, the the yellow color indicates that the related proposition node is true, while the blue color indicates that the related proposition is false. Shown as Fig. 4.4,

After all specifications are composed and all the ROS nodes are pro-

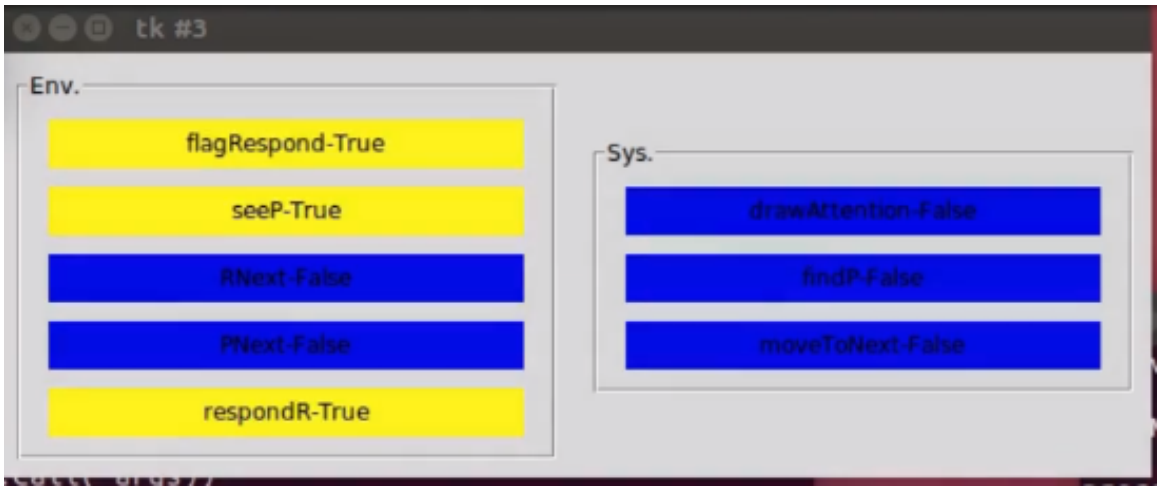

Figure 4.4: An example of the proposition_monitor in LTLstack

grammed, the `setup_launch_file_with_yaml.py` can be used to connect everything together. To use this file, two more documents have to be prepared, `your_project_name.yaml` and `setup_launch_your_project_name.yaml`. `setup_launch_your_project_name.yaml` is used to specify where the *.slugsin* file is, where the ROS nodes are, and where the `your_project_name.yaml` is. `your_project_name.yaml` includes the information about the connection between specific ROS node and its ROS topic. It also contains the information about if the node should be turned on or off when the process is initialized. Replace `your_project_name` to the right project name. How `setup_launch_file_with_yaml.py` is shown in the Fig.4.5,

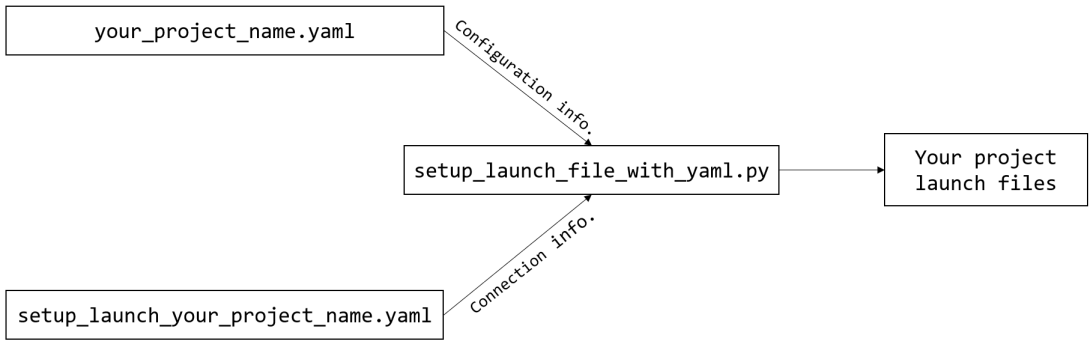

Figure 4.5: A diagram showing how `setup_launch_file_with_yaml.py` works, more details are in Appendix A.

CHAPTER 5

DEMONSTRATION

The functionality of the software framework presented by this thesis is illustrated in the following two examples.

Example 1: Consider a robotic platform is patrolling in a workspace shown in Fig.5.1,

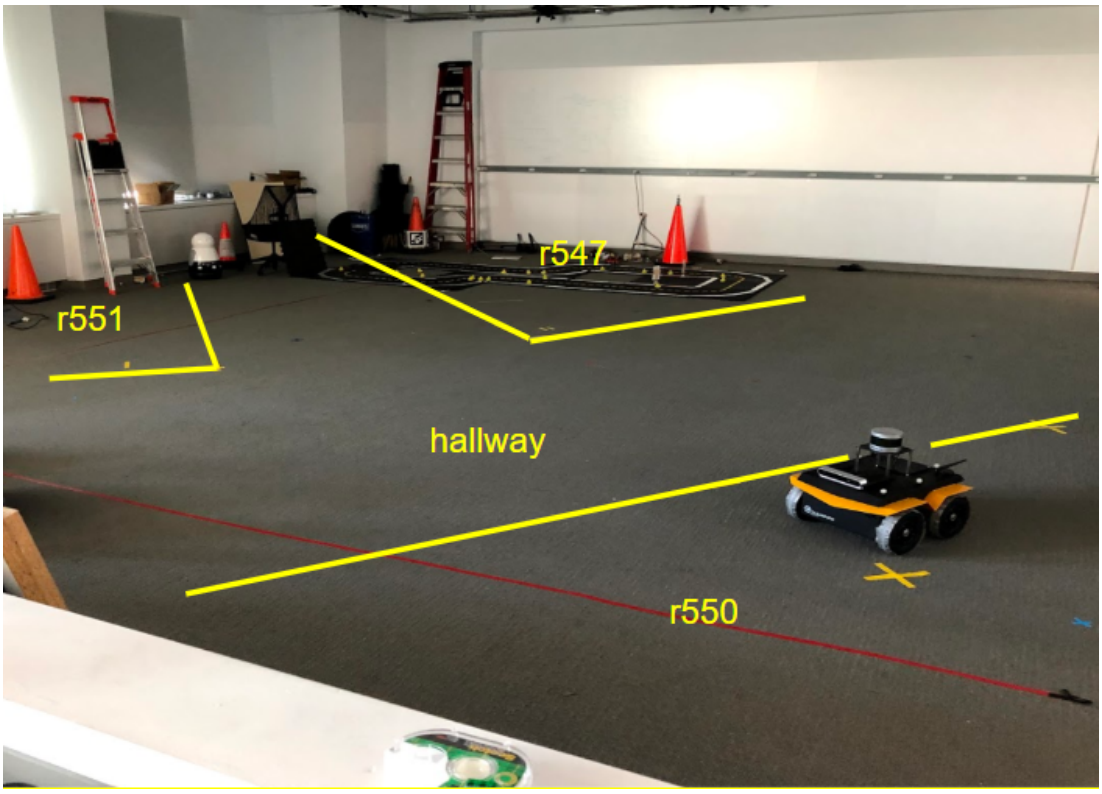

Figure 5.1: The workspace for example 1

which is divided into 4 different sub-workspaces represented by the following propositions $\{r547, r550, r551$ and $hallway\}$ to indicate the different areas. Initially, the robot can be located at any one of these four sub-workspaces, either $r547, r550, r551$ or $hallway$. In nature language, the specification is: *Patrol between*

r547 and r551. If you are sensing a apriltag label0, then go to r550 and stay there. If you are sensing a apriltag label1, then go to back to your original patrolling route.

In this example, there are five inputs and four outputs. The five inputs are if the robot is located at $r547\_rc$, if the robot is located at $r550\_rc$, if the robot is located at $r551\_rc$, if the robot is located at $hallway\_rc$ and if the apriltag detector detected label0 or label1. The outputs are making the robot go to $r547$, go to $r550$, go to $r551$ and go to $hallway$.

The LTL specification of the environment is

$$\varphi_i^e = true, \tag{5.1}$$

the robot has no assumption about where to start.

The LTL specification of the robot is

$$\varphi_i^s = true, \tag{5.2}$$

the robot has no assumption about its initial configuration.

The specifications of φ_t^e describe how the environment changes during the process. For example, if the robot is located at $r547\_rc$, and its goal is to move to $r547\_rc$, it will eventually arrive in $r547\_rc$. If the robot is at $r547\_rc$, then its next location must be in one of the following sub-workspaces, $r547\_rc$ or $hallway\_rc$. Since there is only one robot operated at the same time, the robot can only be

able to observed in one sub-workspace.

$$
\begin{aligned}
\varphi_t^e = G((hallway\_rc)\&(hallway) \rightarrow X(hallway\_rc)) \\
\wedge G((r547\_rc)\&(r547) \rightarrow X(r547\_rc)) \\
\wedge G((r550\_rc)\&(r550) \rightarrow X(r550\_rc)) \\
\wedge G((r551\_rc)\&(r551) \rightarrow X(r551\_rc)) \\
\wedge G((hallway\_rc)\&(r547) \rightarrow X(hallway\_rc)|X(r547\_rc)) \\
\wedge G((hallway\_rc)\&(r550) \rightarrow X(hallway\_rc)|X(r550\_rc)) \\
\wedge G((hallway\_rc)\&(r551) \rightarrow X(hallway\_rc)|X(r551\_rc)) \\
\wedge G((hallway)\&(r547\_rc) \rightarrow X(hallway\_rc)|X(r547\_rc)) \\
\wedge G((hallway)\&(r550\_rc) \rightarrow X(hallway\_rc)|X(r550\_rc)) \\
\wedge G((hallway)\&(r551\_rc) \rightarrow X(hallway\_rc)|X(r551\_rc)) \\
\wedge G(((hallway\_rc)\&!(r547\_rc)\&!(r550\_rc)\&!(r551\_rc))| \\
((r547\_rc)\&!(r550\_rc)\&!(r551\_rc)\&!(hallway\_rc))| \\
((r550\_rc)\&!(r547\_rc)\&!(r551\_rc)\&!(hallway\_rc))| \\
((r551\_rc)\&!(r547\_rc)\&!(r550\_rc)\&!(hallway\_rc)))
\end{aligned}
\tag{5.3}
$$

The specifications of φ_t^s describe the constraints of robot actions. For example, we can tell that if the robot is started from $r547\_rc$, the only possible command it could take is either perform $r547$ or perform *hallway*. If the $r550$ command is given, the behavior will be terminated as the requirement cannot

be performed in the continuous world.

$$\begin{aligned}
\varphi_t^s = G((r550_rc) \rightarrow X(hallway)|X(r550)) \\
\wedge G((r547_rc) \rightarrow X(hallway)|X(r547)) \\
\wedge G((r551_rc) \rightarrow X(hallway)|X(r551)) \\
\wedge G((hallway_rc) \rightarrow X(r547)|X(r550)|X(r551)|X(hallway)) \\
\wedge G((X(r547)\&!X(r550)\&!X(r551)\&!X(hallway))| \\
(X(r550)\&!X(r547)\&!X(r551)\&!X(hallway))| \\
(X(r551)\&!X(r547)\&!X(r550)\&!X(hallway))| \\
(X(hallway)\&!X(r547)\&!X(r550)\&!X(r551)))
\end{aligned} \tag{5.4}$$

The specifications of φ_g^e describe the goal of environment. For example, if the *hallway* command is given to the robot, the robot will eventually arrived at hallway, which changes the environment sensor reading of *hallway_rc*.

$$\begin{aligned}
\varphi_g^e = GF((hallway) \rightarrow (hallway_rc)) \\
\wedge GF((r547) \rightarrow (r547_rc)) \\
\wedge GF((r550) \rightarrow (r550_rc)) \\
\wedge GF((r551) \rightarrow (r551_rc))
\end{aligned} \tag{5.5}$$

Also, as I have additional requirements for the robot, the robot should be able to go to $r550$ while the robot senses apriltag label0. When the robot senses apriltag label1, the robot should repeatedly visit $r547$ and $r551$. These requirements are also composed into the φ_g^s.

$$\begin{aligned}
\varphi_g^s = GF((sensor) \rightarrow (r547_rc)) \\
\wedge GF(!(sensor) \rightarrow (r550_rc)) \\
\wedge GF((sensor) \rightarrow (r551_rc))
\end{aligned} \tag{5.6}$$

Having the model described above, together with

$$\begin{aligned} \varphi^e &= \varphi_i^e \wedge \varphi_t^e \wedge \varphi_g^e \\ \varphi^s &= \varphi_i^s \wedge \varphi_t^s \wedge \varphi_g^s \\ \varphi &= \varphi^e \rightarrow \varphi^s \end{aligned} \tag{5.7}$$

The automaton is built using slugs. The next step is to make the connection between the inputs/outputs and the ROS nodes accordingly. Each pre-made ROS node is connected to one input/output. The ROS node is made as publisher if the node is connected to the output, since the commands, in ROS message, are published through ROS topics. The ROS node is made as subscriber if the node is connected to the input, since the signals, in ROS message, are received through ROS topics. In this example, there are five environment (sensor) ROS nodes and four system (actuator) ROS nodes. The environment (sensor) ROS nodes are used to tell the automaton where the robot is and if the sensor is on. The system (actuator) ROS nodes are used to send velocity commands to the robotic platform following the next valid state calculated based on the automaton.

In this example, AMCL is used to localize the robot. A Lidar sensor is attached to the robot and provides the measurements reflected pulses information. This information is used to compare to the given map to find the position of the robot. In Fig.5.2, the robot is trying to align the sensing information, showing in colorful lines, with the map boundary, showing in black lines to determine its location in the map.

All the goals were met in the test, the project result video screen shots are

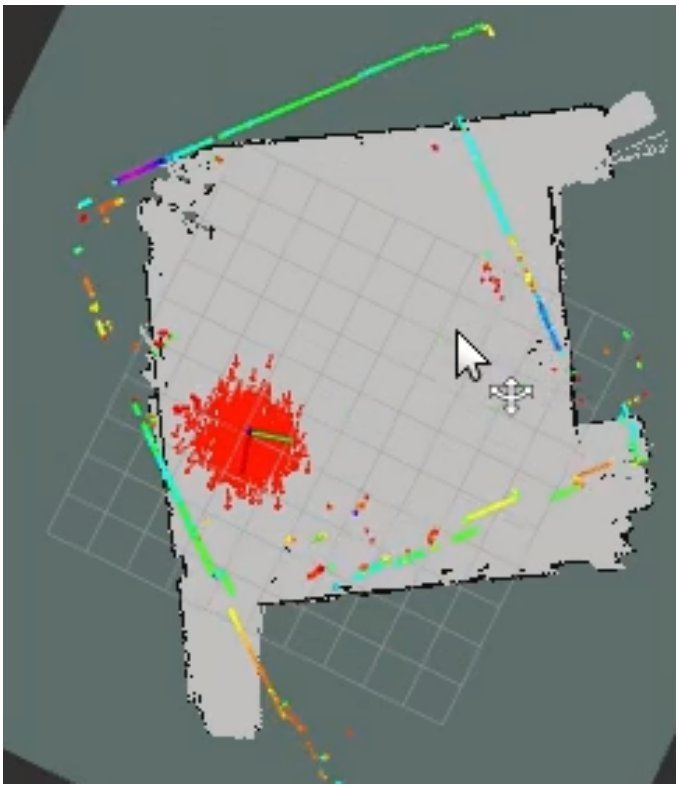

Figure 5.2: AMCL is in the process trying to find the location of the robot relative to the given map

shown as following,

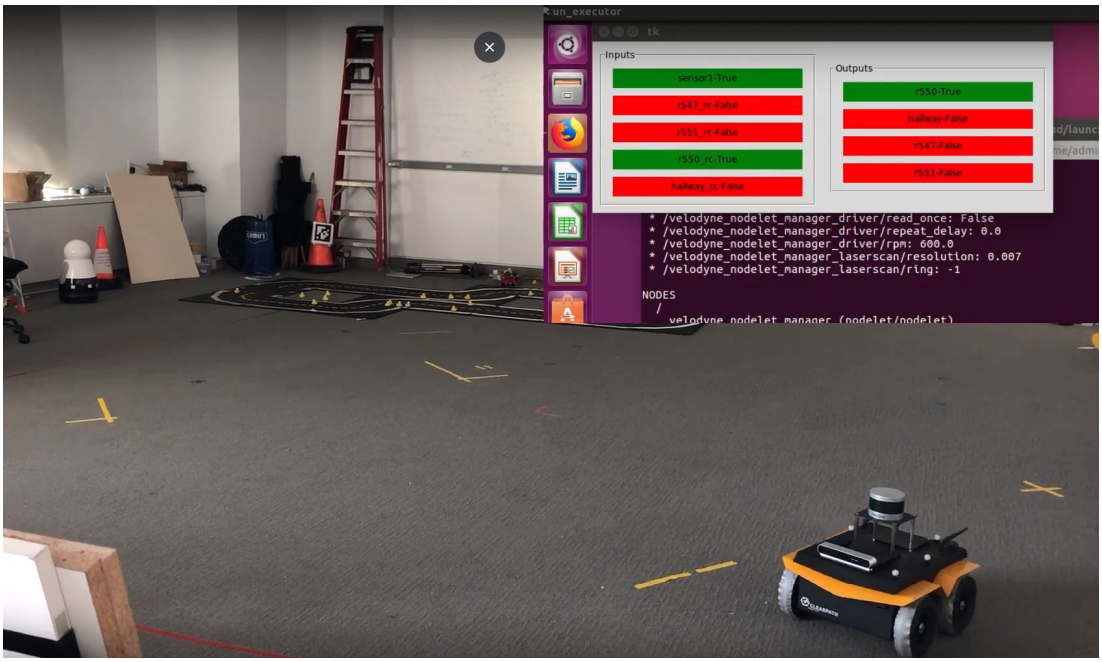

Figure 5.3: The initial state for the robot. The robot was in *r*550, since the apriltag input was true.

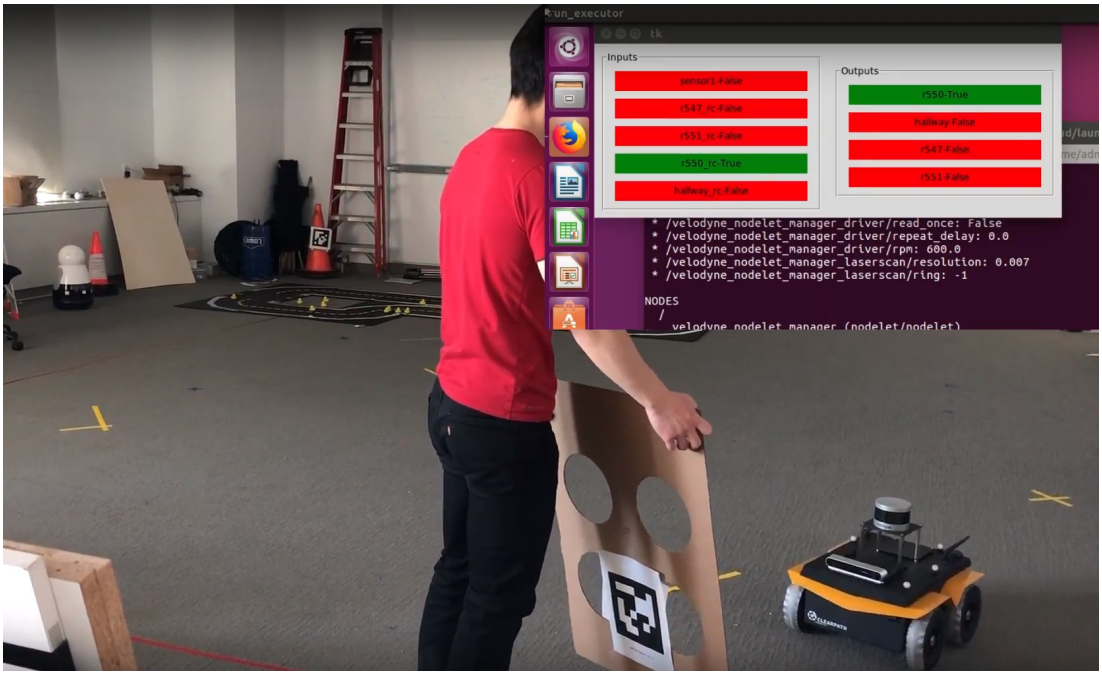

Figure 5.4: The apriltag label1 was shown to the robot, the apriltag sensor was changed to false. The robot was about to get to the next valid sub-workspace.

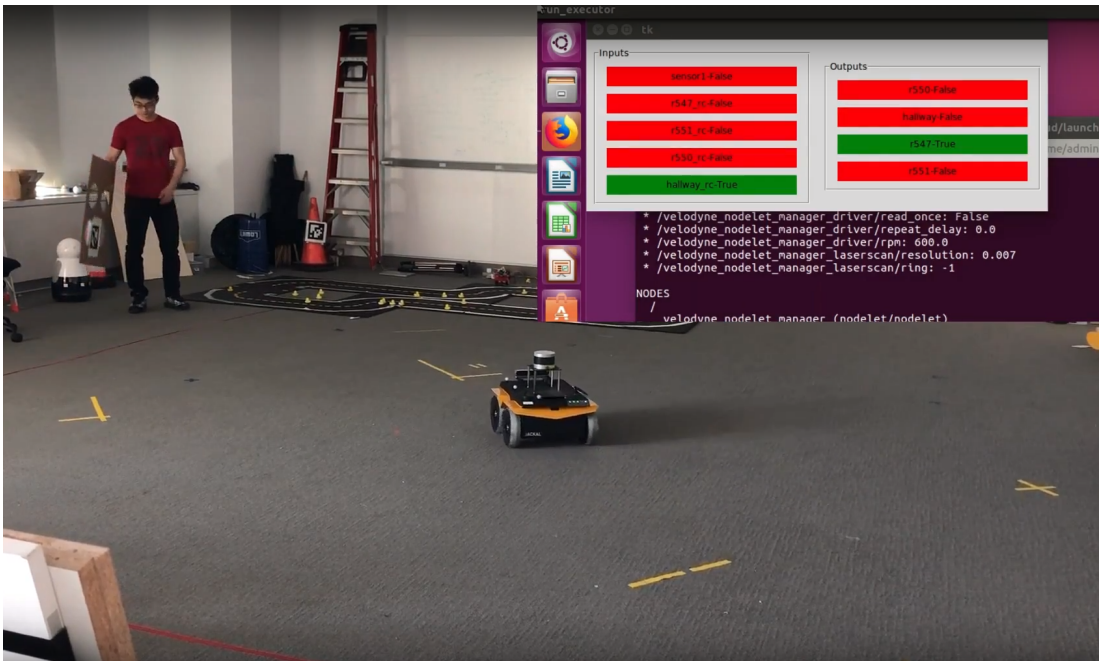

Figure 5.5: After the apriltag sensor was changed to false, the robot was heading towards *r*547_*rc*.

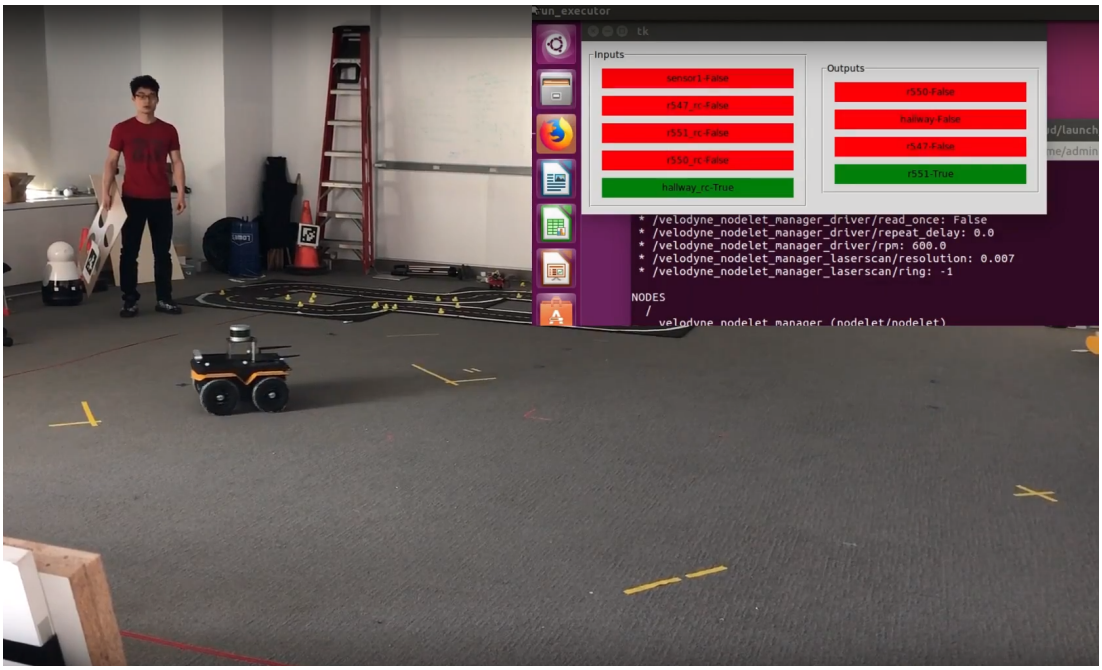

Figure 5.6: After *r*547_*rc* had been visited, the robot was heading to *r*550_*rc* to complete the task of patrolling between *r*547_*rc* and *r*550_*rc*.

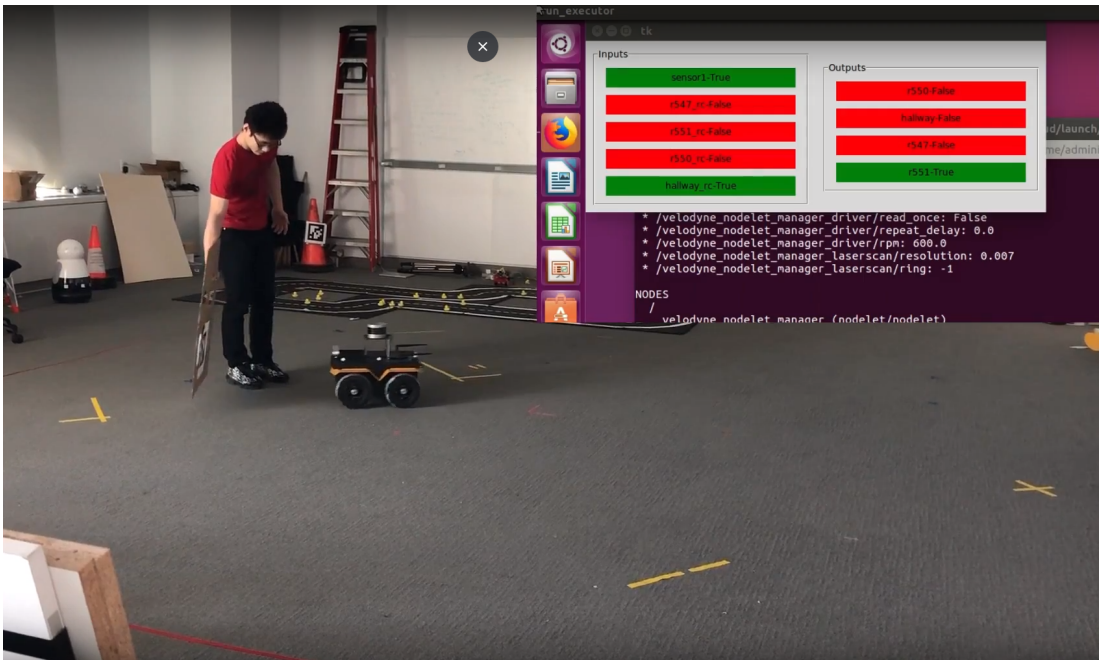

Figure 5.7: The apriltag label0 was shown to the robot, the apriltag sensor was changed to on. The robot was about to get to the next valid sub-workspace.

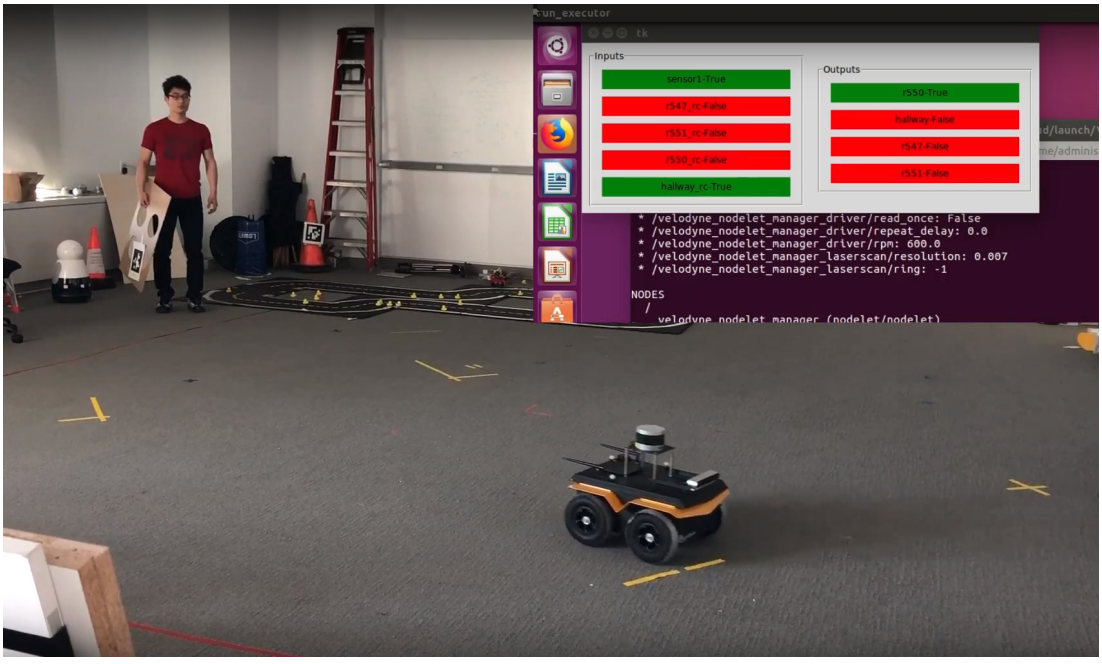

Figure 5.8: After the apriltag sensor was changed to true, the robot was heading towards *r*550_*rc*.

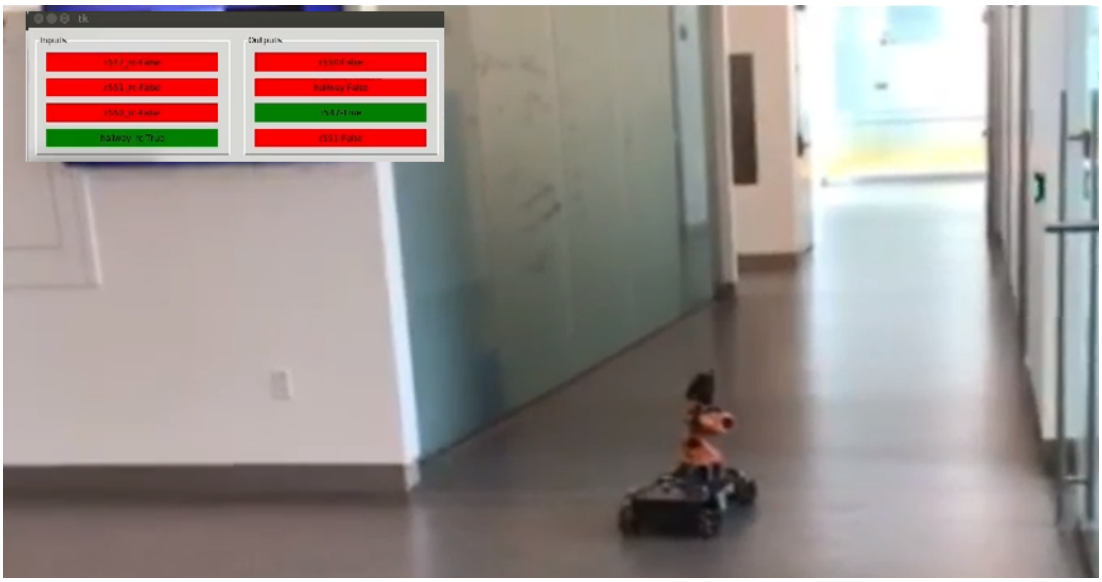

Figure 5.9: KUKA youbot was performing tasks using LTLstack in the hallway. The robot uses AMCL to localize its position in the hallway. The specification of this task is very similar to example 1, while the map is changed to hallway, a non-Vicon environment.

Example 2: Consider a robotic platform is performing a specific task in a workspace shown in Fig.5.10,

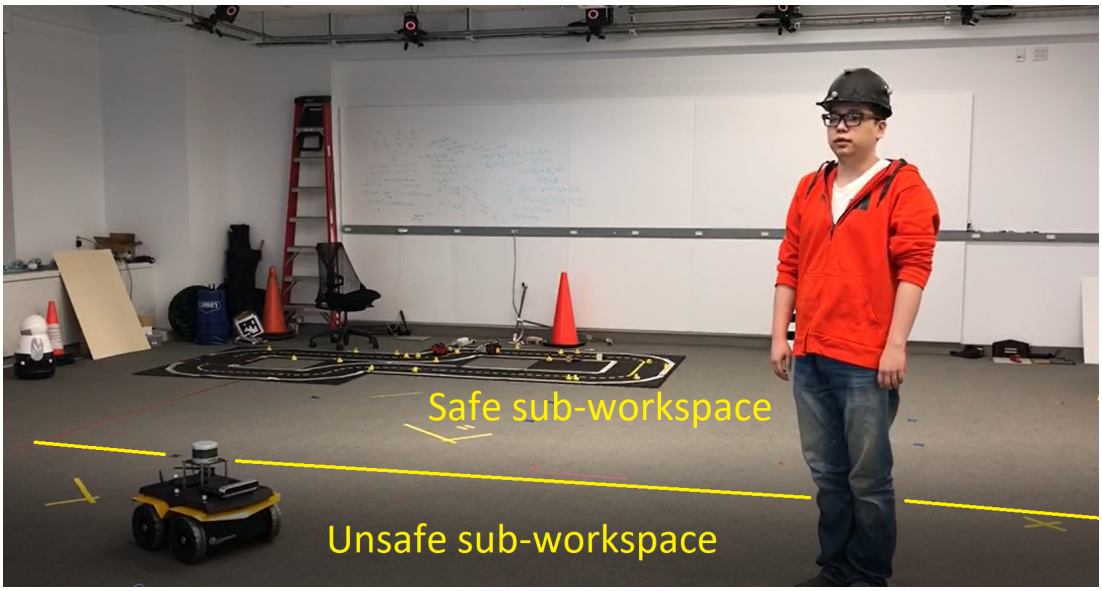

Figure 5.10: The workspace for example 2

which is divided into 2 different sub-workspaces represented by the following propositions $\{W_0, W_1\}$. There is a person and a robot in the specifications. We define W_0 to be the unsafe sub-workspace. Imagine some bad things are happening in this sub-workspace. We define W_1 to be the safe sub-workspace, which is the sub-workspace we want the person and the robot to be in. We assume the person is willing to cooperate with the robot. We also assume both person and robot are located in the unsafe sub-workspace initially. In nature language, the specification is: *Search for the person in the unsafe sub-workspace while showing the exit information. If you find the person, wait for the person to give you a signal showing the person read the information. Go to the safe sub-workspace, after the person is already in the safe sub-workspace. If the person gave you the signal and still wondered in the unsafe sub-workspaces, keep finding the person and repeat the process.*

In this example, there are five inputs and three outputs. The five inputs are if the person is located in safe sub-workspace, $PNext$, if the robot is located in safe sub-workspace, $RNext$, if the robot is getting respond from the person, $respondR$, if the robot has the person in its sight, $seeP$, and if the robot has ever get the respond from the person, $flagRespond$. The outputs are have the robot to draw attention from the person, $drawAttention$, move to the safe sub-subspace, $moveToNext$ and perform the action of finding the person $findP$.

The LTL specification of the environment is

$$\varphi_i^s = true, \tag{5.8}$$

the robot has no assumption about its initial configuration.

The LTL specification of the robot is

$$\varphi_i^e = !PNext \wedge !RNext \wedge !respondR \wedge !seeP \wedge !flagRespond, \tag{5.9}$$

we assume both the person and the robot are in the unsafe sub-workspace. We also assume at beginning of the process, the robot didn't find the person yet and the person didn't respond to the robot yet.

The specifications of φ_t^e describe how the environment changes during the process. For example, if the robot is in the safe sub-workspace, it will always be there. Same situation applies to the person. We assume only if the robot sees the person, the person have the ability to respond to the robot.

$$\begin{aligned}
\varphi_t^e = G((RNext) \rightarrow X(RNext)) \\
\wedge G((PNext) \rightarrow X(PNext)) \\
\wedge G(!(PNext) \rightarrow !X(RNext)) \\
\wedge G(!(seeP) \rightarrow !X(respondR)) \\
\wedge G((respondR) \rightarrow X(seeP)) \\
\wedge G((!(flagRespond)\&!(respondR)) \rightarrow !X(flagRespond)) \\
\wedge G((!(flagRespond)\&(respondR)) \rightarrow X(flagRespond)) \\
\wedge G((flagRespond) \rightarrow X(flagRespond))
\end{aligned} \tag{5.10}$$

The specifications of φ_t^s describe the constraints of robot actions. For example, if the person is not found, the person is not in the safe sub-workspace and the person didn't respond to the robot yet, then the *drawAttention* and $findP$ should be turned on. If the person is already in the safe sub-workspace, then

the robot should move to the safe sub-workspace.

$$\begin{aligned}\varphi_t^s = G(!(seeP)\&!(respondR)\&!(PNext) \rightarrow X(drawAttention)\&X(findP)\&!X(moveToNext))\\ \wedge G((seeP)\&!(respondR)\&!(PNext) \rightarrow X(drawAttention)\&!X(findP)\&!X(moveToNext))\\ \wedge G((respondR)\&!(PNext) \rightarrow !X(drawAttention)\&!X(findP))\\ \wedge G((PNext) \rightarrow X(moveToNext))\\ \wedge G(!(PNext) \rightarrow !X(moveToNext))\\ \wedge G(!((drawAttention)\&(moveToNext)))\end{aligned} \tag{5.11}$$

The environment has the following assumptions on liveness, such as if the robot continuously trying to find the person, it will eventually find the person. If the robot find the person and *drawAttention* is on, the person will eventually respond to the robot.

$$\begin{aligned}\varphi_g^e = GF((findP) \rightarrow (seeP))\\ \wedge GF((seeP)\&(drawAttention) \rightarrow (respondR))\\ \wedge GF((flagRespond) \rightarrow (PNext))\\ \wedge GF((moveToNext) \rightarrow (RNext))\\ \wedge GF(!(respondR) \rightarrow (PNext))\end{aligned} \tag{5.12}$$

Also, as I have additional requirements for the robot and the person, they should be eventually in the safe sub-workspace at the same time.

$$\varphi_g^s = GF((PNext)\&(RNext)). \tag{5.13}$$

Having the model described above, together with

$$\begin{aligned} \varphi^e &= \varphi_i^e \wedge \varphi_t^e \wedge \varphi_g^e \\ \varphi^s &= \varphi_i^s \wedge \varphi_t^s \wedge \varphi_g^s \\ \varphi &= \varphi^e \rightarrow \varphi^s \end{aligned} \tag{5.14}$$

the automaton is built using slugs, the next step is to make the connection between the inputs/outputs and the ROS nodes accordingly. Each pre-made ROS node is connected to one input/output. The ROS node is made as publisher if the node is connected to the output as the commands, in ROS message, are published through ROS topics. The ROS node is made as subscriber if the node is connected to the input as the signals, in ROS message, are received through ROS topics. In this example, there are five environment (sensor) ROS nodes and three system (actuator) ROS nodes. The environment (sensor) ROS nodes are used to tell the automaton where the robot is and whether the person is found or not. The system (actuator) ROS nodes are used to send velocity commands to the robotic platform following the next valid state calculated based on the automaton. In this example, the velocity commands are calculated through certain dynamics to fulfill the specific requirements. The velocity commands can be send to any receiver-based channel selection for wireless networks. In the test, we sent the velocity commands via TCP protocol, and the commands were received and processed by the robot.

All the goals were met in the test, the project pictures are shown as following,

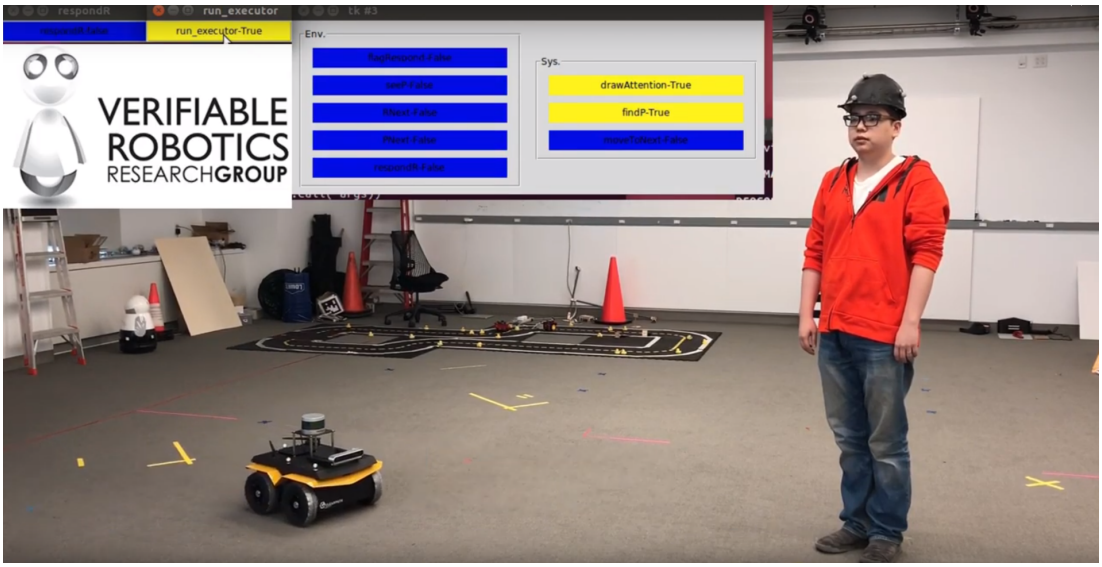

Figure 5.11: The initial state for the robot. Both the robot and the person were in unsafe sub-workspace. The robot was trying to find the person.

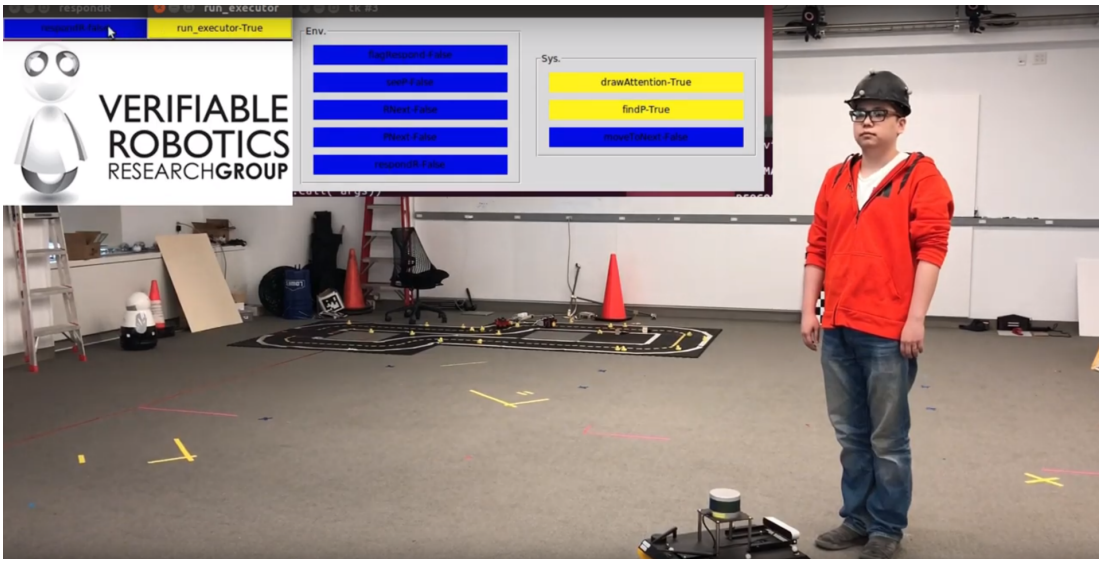

Figure 5.12: The robot had found the person. And then, it was trying to see the person.

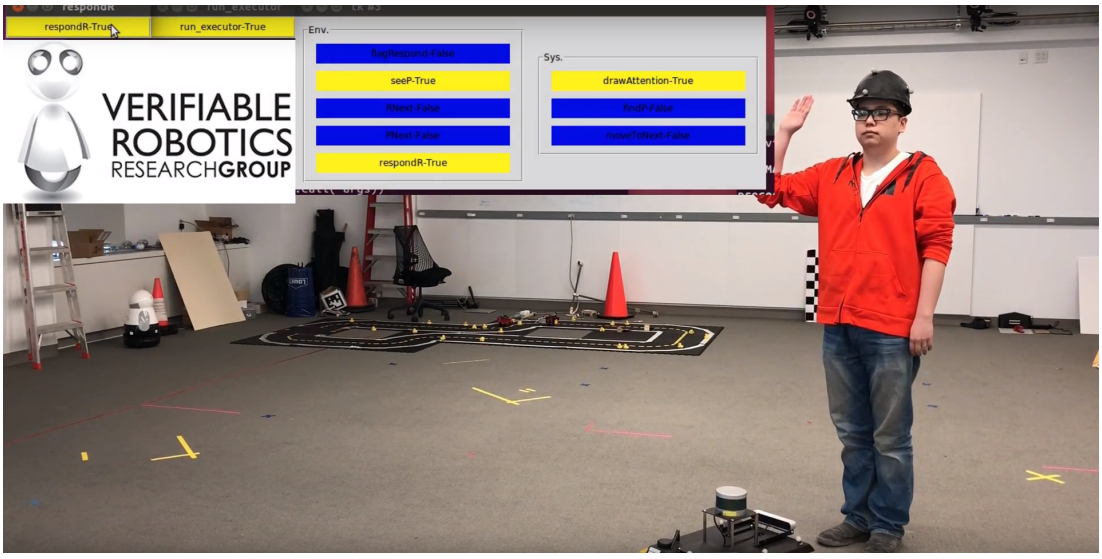

Figure 5.13: After the robot saw the person, the person responded to the robot.

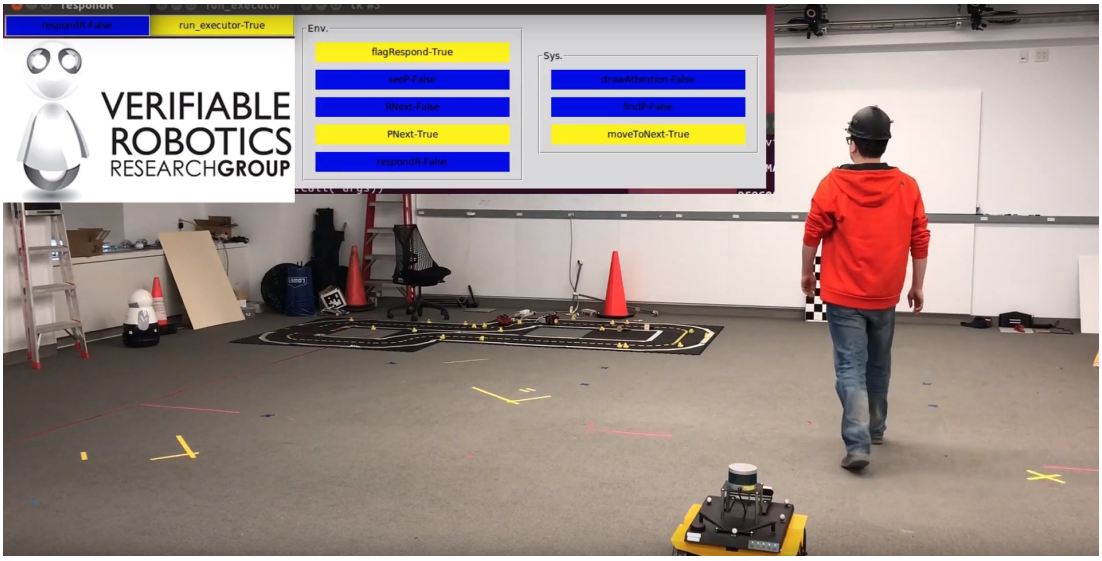

Figure 5.14: Case 1, After the person responded to the robot, the person went to the safe sub-workspace.

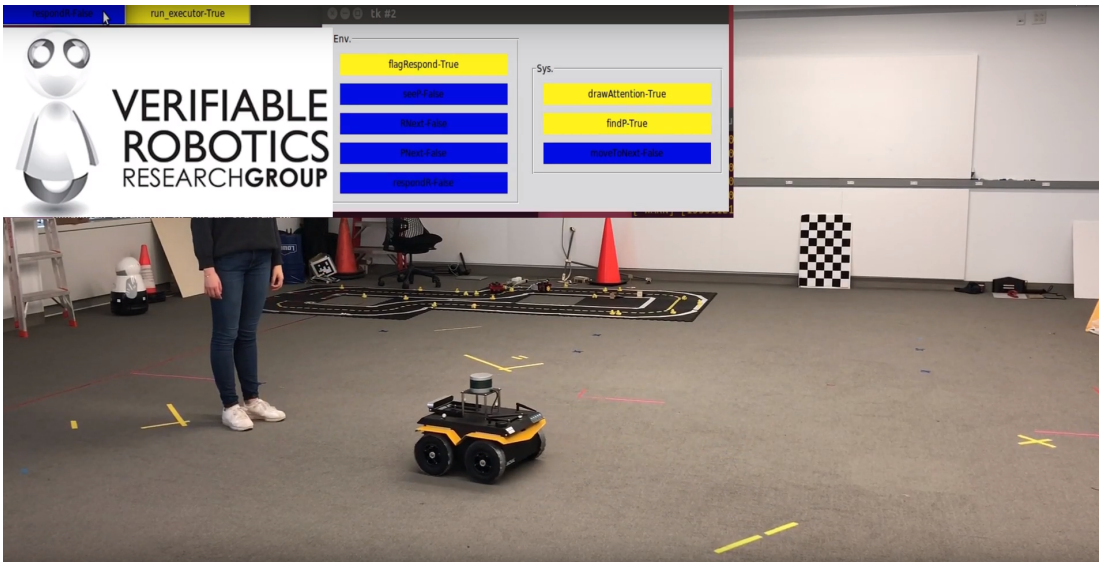

Figure 5.15: Case 2, After the person responded to the robot, the person still wondered in the unsafe sub-workspace. The robot was trying to find the person again.

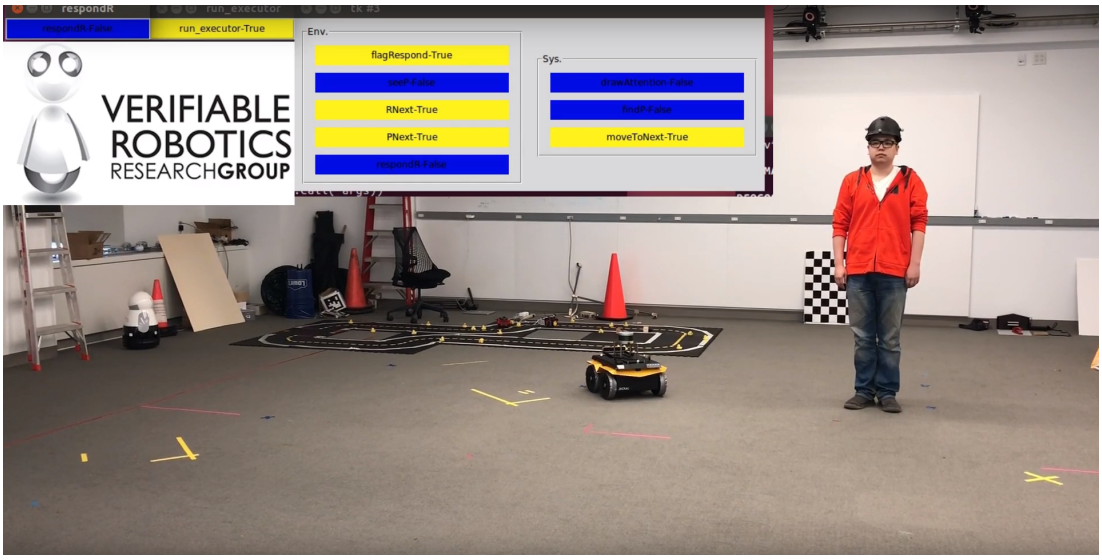

Figure 5.16: After the person was in the safe sub-workspace. The person and the robot were in the safe sub-workspace at the same time eventually.

APPENDIX A

APPENDIX: USER'S GUIDE

1. System Requirements:

   LTLstack can be used under the environment of Ubuntu 14.04 with ROS Indigo installed or Ubuntu 16.04 with ROS Kinetic installed.

2. Versions:

   LTLstack-Indigo can be installed under Ubuntu 14.04.

   `https://github.com/VerifiableRobotics/LTL_stack/tree/master`

   LTLstack-Kinetic can be installed under Ubuntu 16.04.

   `https://github.com/VerifiableRobotics/LTL_stack/tree/ros-kinetic`

3. Dependencies:

   The following dependencies are required: 1.slugs 2.Tkinter

   How to install slugs:

   (a) Install the specific version of slugs: `git clone -b LTL_stack https://git@github.com/wongkaiweng/slugs.git`

   (b) Install the dependence: `sudo apt-get install libboost-all-dev`

   (c) Compile slugs by running `make` in the *slugs*/*src* directory

   (d) Make sure slugs can be found anywhere by adding this line `export PATH=:$PATH:</path/to/slugs-src-folder>` to your ~ /*.bashrc* file.

   How to install Tkinter: `sudo apt-get install python-tk`

4. Try an example:

   (a) Download the repo into your workspace *src* folder: `git clone https://github.com/VerifiableRobotics/LTL_stack.git`. After the master branch is downloaded, pull the `ros-kinetic` branch to have the latest version.

   (b) Find the directory to your workspace and run: `catkin_make`

   (c) Check if all the LTLstack nodes are executable. If not, run `find <dir to LTL_stack> -type f -exec chmod 755 {} \;`. Replace the `<dir to LTL_stack>` to your LTLstack folder.

   (d) Try to run the Rotating Turtle example! And have fun!

   run `roslaunch controller_executor tutorial_all.launch`

5. How to write a proposition node:

   There are two main kinds of ROS node, the publisher node and the subscriber node. In LTLstack, a publisher node, aka. system (actuator) node, is a node sending messages to the robotic platform, while a subscriber node, aka. environment (sensor) node, is a node receiving messages from the environment sensors.

   (a) Publisher (system) node:

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String


def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
```

```
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

According to the publisher node example above, let's read in detail to learn how the node is made.

```
#!/usr/bin/env python
```

Every Python ROS node should have this declaration at the top. The first line makes sure your script is executed as a Python script.

```
import rospy
from std_msgs.msg import String
```

You need to import `rospy` if you are writing a ROS node. The *std_msgs.msg* contains the basic message type we can use, and in this example, the `String` is used.

```
pub = rospy.Publisher('chatter', String, queue_size=10)
rospy.init_node('talker', anonymous=True)
```

This section of the code defines the publisher, including which topic to publish, the message type in the topic, and how many pieces of message can be in the topic at the same time. The second line defines

the node's name. When your node's name is taken by previous ROS nodes, this section `anonymous=True` can help you to unique your ROS node by adding random number to the end of the node's name.

```
rate = rospy.Rate(10) # 10hz
```

This line helps you to define in which frequency you want your message to be published.

```
while not rospy.is_shutdown():
    hello_str = "hello world %s" % rospy.get_time()
    rospy.loginfo(hello_str)
    pub.publish(hello_str)
    rate.sleep()
```

This loop keeps the node to continuously publish message until the *roscore* is shut down. `rate.sleep()` has very similar function as `time.sleep()` in python.

```
if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

The main block triggers the function, `talker()`, and try to catch the exception.

(b) Subscriber (environment) node:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
```

```
        rospy.loginfo(rospy.get_caller_id() + "I heard %s"
        ..., data.data)

    def listener():
        rospy.init_node('listener', anonymous=True)
        rospy.Subscriber("chatter", String, callback)
        rospy.spin()

if __name__ == '__main__':
        listener()
```

The subscriber node is very similar to the publisher node. The biggest difference is the additional `callback` function, which is not shown in the publisher node.

```
        rospy.init_node('listener', anonymous=True)
        rospy.Subscriber("chatter", String, callback)
        rospy.spin()
```

This section of the node initializes the node and "give" the node its role, the subscriber. The node subscribes the message from the topic `chatter` in the type of `String`. The message is coming from the `callback` which is defined in the following section.

```
    def callback(data):
        rospy.loginfo(rospy.get_caller_id() + "I heard %s"
        ..., data.data)
```

This is a python function, inside this function is how the message is generated. We can also make another subscriber node here to subscribe to another topic. In this way, we can make a connection among three or more nodes.

6. Files needed to make a LTLstack package:

   (a) A specification in *.slugsin* format. In this thesis, the specification can be generated by LTLMoP.

   (b) A YAML file that maps the ROS proposition nodes with the inputs/outputs of the specification. Below is a YAML file from example 1 in the thesis.

```yaml
inputs:
  sensor1:
    node : 'sensor1'
    node_publish_topic : '/test_806/inputs/sensor1'
    pkg : 'controller_executor'
    filename : 'tk_button.py'
    parameters :
      init_value: false

  hallway_rc:
    node : 'hallway_rc'
    node_publish_topic : '/test_806/inputs/hallway_rc'
    pkg : 'controller_executor'
    filename : 'test_806_hallway_rc.py'
```

Figure A.1: An example of the YAML file in LTLstack

   (c) Create a *SetupLaunch* YAML File. This is the file to specify where the *.slugsin* file is, where all the ROS nodes are and where the previous YAML file is. Below is a YAML file from example 1 in the thesis.

7. How to create a new example:

   (a) Create environment (sensor)/system (actuator) ROS nodes following the instruction of Appendix A.5

```
slugsin_file : '/home/chuanwei/ms_project/test_806/test_806.slugsin'
region_file : None
destination_folder: '/home/chuanwei/catkin_ws/src/LTL_stack/controller_executor/examples/test_806'
example_name : 'test_806'
yaml_file : '/home/chuanwei/catkin_ws/src/LTL_stack/controller_executor/examples/test_806/test_806.yaml'
LTLMoP_src_dir : '/home/chuanwei/LTLMoP/src'
controller_executor_dir : '/home/chuanwei/catkin_ws/src/LTL_stack/controller_executor'
init_region : ''
rot : 0
scale : 0.01
x_trans : 0
y_trans : 0
```

Figure A.2: An example of the setup YAML file in LTLstack

(b) Create the Proposition YAML File following the instruction of Appendix A.6

(c) Create the setup YAML File following the instruction of Appendix A.6

(d) With the two YAML files, you can automatically generate all the launch files with the following command:

`rosrun controller_executor setup_launch_file_with_yaml.py [setup_launch_yaml_file_dir/setup_launch_yaml_file_name]`

8. How to run a new example:

After all the previous steps, running an example is straight forward. Just run: `roslaunch controller_executor <your example name>_all.launch`. Replace the `<your example name>` with your example name.

APPENDIX B

APPENDIX: DEBUG A LTLSTACK PACKAGE

To debug a LTLstack package, we can use the following decision tree, fig B.1 as guide to find and fix the bugs.

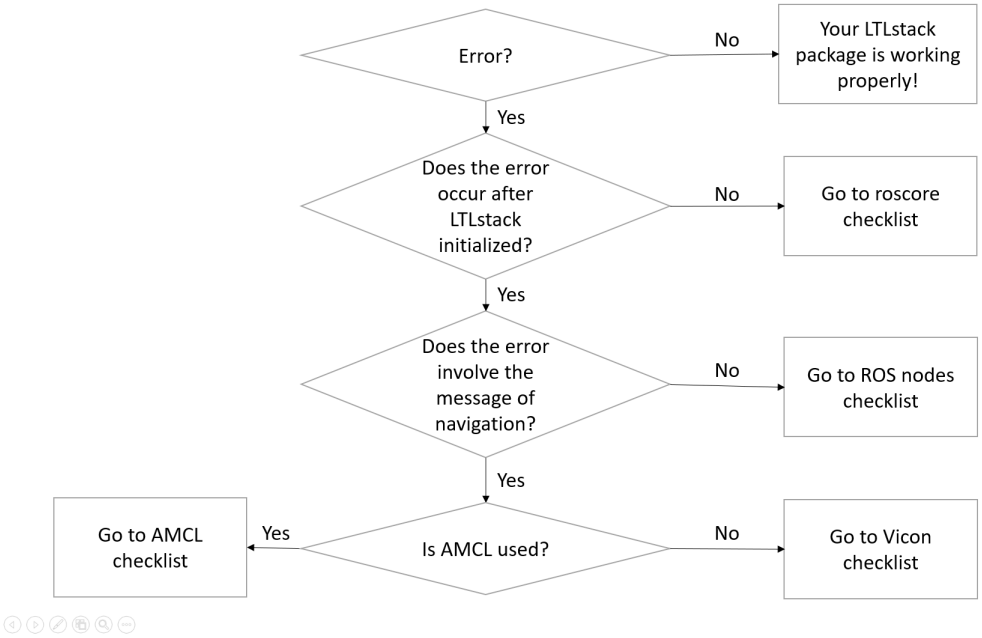

Figure B.1: The decision tree for the debug process for LTLstack package

`roscore` Checklist:

1. Is another computer in the same wifi environment running the `roscore`?

   If so, turn that terminal off to make sure your operating computer is the only one running `roscore` at the time.

2. Is the `ROS_MASTER_URI` setting right in `/.bashrc`?

   The `/.bashrc` interprets your typed input in the Terminal program and runs commands based on your input. To trigger the right `ROS_MASTER_URI` while `roscore` is called, you should

insert `export ROS_MASTER_URI=http://<the IP address where the roscore should be running on>:11311/` in `/.bashrc`. Change `<the IP address where the roscore should be running on>` to your desired IP address. Don't forget to restart all the terminals after the `/.bashrc` is changed, since the changes wouldn't affect the existing terminals.

ROS node checklist:

Go to `<your ROS nodes dir>`, run `python <the related ROS node>.py` to debug the related ROS node based on the information provided via python. Change `<your ROS nodes dir>` to where the ROS nodes are stored. Change `<the related ROS node>` to the ROS node you wanted to analyze.

AMCL checklist:

1. Is another computer in the same wifi environment running the AMCL package?

   If so, turn that terminal off to make sure your operating computer is the only one running `roscore` at the time.

2. Is the AMCL package turned on?

   If not, run the AMCL package to trigger the package. The AMCL package vary depends on different robotic platform is used. For example, if Jackal is used, you would run `roslaunch jackal_navigation amcl_demo.launch map_file:=<path to your map>.yaml`. Change `<path to your map>` to your map.

3. Are the related AMCL topics established properly?

   To check this, run `rostopic list` to see if the desired AMCL topics are listed.

4. Are the related AMCL topics publishing the proper message?

   To check this, run `rostopic echo <the topic you want to analyze>` to see if the vicon topic was publishing the proper message. Change `<the topic you want to analyze>` to your ROS AMCL topic. Here are some commonly used AMCL topics: `amcl_pose`, robot's estimated pose in the map; `particlecloud`, the set of pose estimates being maintained by the filter; `tf`, publishes the transform from odometry to map.

Vicon checklist:

1. Is another computer in the same wifi environment running the `vicon_bridge` package?

   If so, turn that terminal off to make sure your operating computer is the only one running `roscore` at the time.

2. Is the vicon package turned on?

   If not, run `roslaunch vicon_bridge vicon.launch` to trigger the package.

3. Are the desired vicon objects checked?

   If not, check the related vicon objects in the vicon system, then click the "track" button in the vicon system.

4. Are the related vicon topics established properly?

   To check this, run `rostopic list` to see if the desired vicon topics are listed.

5. Are the related vicon topics publishing the proper message?

   To check this, run `rostopic echo <the topic you want to analyze>` to see if the vicon topic is publishing the proper message. Change `<the topic you want to analyze>` to your ROS vicon topic.

6. Is the transform package properly used?

   To check this, run `rosrun tf view_frames`. This would provide you the information about how relationship between coordinate frames is made in the ROS system. Usually, if the above 5 check points don't give you any unexpected result, there are problems hidden in the transform relationship related to vicon.

APPENDIX C

APPENDIX: DEVELOPER'S GUIDE

In this section, the software introduced in this thesis is analyzed in detail. This section is designed for developers who are interested in improving and developing the later version of LTLstack.

1. Files in a LTLstack example package

   To understand how LTLstack works, we have to know what is in an example package. Following files are included in an example package,

   (a) *setup_launch_tutorial.yaml*

   This is the configure file to automatically generate all the launch files.

   (b) *tutorial.slugsin*

   This is the LTL specification in slugs format.

   (c) *tutorial.yaml*

   This defines how each proposition corresponds to a ROS file. It also specifies the topic of each proposition.

   (d) *tutorial_background.launch*

   This launches nodes other than proposition nodes.

   (e) *tutorial_executor.launch*

   This takes in the LTL specification, synthesize a controller and execute it with slugs.

   (f) *tutorial_propositions.launch*

   This launches all the proposition files as defined in *tutorial_propositions.launch*.

(g) *tutorial_all.launch*

This launches all the other launch files in the folder. An easy way to start running the example.

All the files above can be manually modified, so we can manipulate the structure of LTLstack accordingly. For example, we can use one location sensor ROS node to replace several redundant ROS nodes, then the processing speed can be improved to avoid possible delay related to calculation power.

2. Change synthesizer option in slugs

Slugs has several different synthesizer options to make the automaton. Each option has its own benefits and disadvantages. To change the option, the developer should:

(a) Edit the file named *slugs_startup_server_base.py*. In this file, the developer can find a class called `SlugsExecutorBase`.

(b) Find `Initialize Slugs Execution Service`, `Slugs Transition Service` and `Slugs Goal Service` in `SlugsExecutorBase`. They are all initially linked to the ROS server based on the interactive option. These three sub-sections make the main body of the ROS server, and the automaton building option here is the interactive option.

(c) If we changed the interactive option to *explicitStrategy*, we have to store the explicit-states, the outputs from *explicitStrategy*, in a reference file, and link this file to these three sub-sections accordingly. The next valid state is not a result generated from the automaton directly anymore, it will be a result from the reference file based on the related

input.

3. *input_manager* structure

   The *input_manager* is made to deal with the environment sensor information. As the code shown below,

```
#! /usr/bin/env python
import argparse
import rospy
import yaml
import logging
import time
import std_msgs.msg

import controller_executor.msg

import file_operations
import controller_executor_logging
input_manager_logger = logging.getLogger("input_manager_logger")

##### EXAMPLES ##
# roscore
# rosrun turtlesim turtlesim_node
#  rosrun turtlesim turtlesim_node __name:=turtle2 /turtle1:=/turtle2

# rosnode cleanup

#rosservice call /turtle1/set_pen 200 0 0 2 0

# python src/input_manager.py examples/simple.yaml
########################
```

```
input_readings = {}

class InputManager(object):
    _input_readings = {}

    def __init__(self, yaml_file, node_name):
        # load yaml file
        input_prop_to_ros_info, output_prop_to_ros_info = file_operations.load

        # subscribe to topics
        self.subscribe_to_topics(input_prop_to_ros_info)

        # setup publisher for inputs dict
        self.input_dict_pub = rospy.Publisher(node_name+'/incoming_inputs',
        ... controller_executor.msg.stringKeyBoolValueDict, queue_size=10)
        input_manager_logger.info("Finished Input Manager Initialization...")

    def update_value(self, data, prop):
        if not self._input_readings[prop] == data.data:
            input_manager_logger.debug('Prop-{0}:{1}'.format(prop, data.data))

        self._input_readings[prop] = data.data
        rospy.loginfo(self._input_readings)

    def subscribe_to_topics(self, prop_to_ros_info):
        for prop, prop_info in prop_to_ros_info.iteritems():
            self._input_readings[prop] = False
            rospy.Subscriber(prop_info['node_publish_topic'],
            ... std_msgs.msg.Bool,
            ... callback=self.update_value, callback_args=prop)

if __name__ == '__main__':
```

```
parser = argparse.ArgumentParser(description="Input Manager")
parser.add_argument('yaml_file', type=str, help='Specify .yaml
... file directory')
#parser.add_argument('--input_manager_name', type=str,
... help='Specify name of the input manager.',\
nargs='?', const='input_manager', default='input_manager')

args, unknown = parser.parse_known_args()

# initialize node
rospy.init_node('input_manager') #args.input_manager_name)

# initialize manager
manager = InputManager(args.yaml_file, 'input_manager')
... #args.input_manager_name)

# publish all collected sensors
input_manager_logger.info("Start publishing inputs ...")

rate = rospy.Rate(10)
while not rospy.is_shutdown():
    # form message
    inputs = controller_executor.msg.stringKeyBoolValueDict
    ... (manager._input_readings.keys(), manager._input_readings.values())

    # publish
    manager.input_dict_pub.publish(inputs)
    #input_manager_logger.debug("!!!444444publish inputs"+str(inputs))

    # wait
    rate.sleep()
```

The `subscribe_to_topics` is to gather the sensor information as shown below,

```
def subscribe_to_topics(self, prop_to_ros_info):
        for prop, prop_info in prop_to_ros_info.iteritems():
            self._input_readings[prop] = False
            rospy.Subscriber(prop_info['node_publish_topic'],
            ... std_msgs.msg.Bool,
            ... callback=self.update_value, callback_args=prop)
```

The `update_value` is to update the gathered sensor information and store it in the message type designed for LTLstack,

```
    def update_value(self, data, prop):
        if not self._input_readings[prop] == data.data:
            input_manager_logger.debug('Prop-{0}:{1}'.format(prop, data.data))

        self._input_readings[prop] = data.data
        rospy.loginfo(self._input_readings)
```

Any modification related to sensor information can be realized if the two functions above are changed properly.

4. *output_manager* structure

   The *output_manager* is to send command message to the robotic platform. As the code shown below,

```
#! /usr/bin/env python
import argparse
import rospy
import yaml
import time
import logging
import std_msgs.msg
```

```
import controller_executor.msg

import file_operations
import controller_executor_logging
output_manager_logger = logging.getLogger("output_manager_logger")

class OutputManager(object):
    _output_publishers = {}
    _output_dict = {}

    def __init__(self, yaml_file):

        # load yaml file
        input_prop_to_ros_info, output_prop_to_ros_info =
        ... file_operations.loadYAMLFile(args.yaml_file)

        # set up ros publishers
        self.setup_publish_topics(output_prop_to_ros_info)

        # subscribe to controller (need to first create the manager)
        rospy.Subscriber('executor/incoming_outputs',
        ... controller_executor.msg.stringKeyBoolValueDict,
        ... callback=self.update_outputs)

    def update_outputs(self, data):
        # print new info
        if dict(zip(data.keys, data.values)) != self._output_dict:
            output_manager_logger.debug('Publishing outputs:{0}'.format
            ... (dict(zip(data.keys, data.values))))
            self._output_dict =dict(zip(data.keys, data.values))
```

```
            for idx, prop in enumerate(data.keys):
                for pub in self._output_publishers[prop]:
                    pub.publish(data.values[idx])
                #output_manager_logger.debug('OUTPUT prop {0}:{1}'.format
                ... (prop, data.values[idx]))

        def setup_publish_topics(self, prop_to_ros_info):
            for prop, prop_info in prop_to_ros_info.iteritems():
                if not prop in self._output_publishers.keys():
                    self._output_publishers[prop] = []
                if isinstance(prop_info, list):
                    for prop_info_element in prop_info:
                        self._output_publishers[prop].append(rospy.Publisher
                        ... (prop_info[0]['node_subscribe_topic'],
                        ... std_msgs.msg.Bool,
                        ... queue_size=10))
                else:
                    self._output_publishers[prop].append(rospy.Publisher
                    ...(prop_info['node_subscribe_topic'], std_msgs.msg.Bool,
                    ... queue_size=10))


    if __name__ == '__main__':
        parser = argparse.ArgumentParser(description="Launch output manager")
        parser.add_argument('yaml_file', type=str, help='Path to yaml file')

        args, unknown = parser.parse_known_args()

        rospy.init_node('output_manager')

        output_manager = OutputManager(args.yaml_file)
```

```
rospy.spin()
```

The update_outputs is to update command message, it is also used to determine which message to go where.

Based on the structure of the message designed for LTLstack, there are several parallel propositions and several parallel Boolean message. This function assures the right proposition is linked to the right Boolean message.

```
def update_outputs(self, data):
    # print new info
    if dict(zip(data.keys, data.values)) != self._output_dict:
        output_manager_logger.debug('Publishing outputs:{0}'.format
        ... (dict(zip(data.keys, data.values))))
        self._output_dict =dict(zip(data.keys, data.values))

    for idx, prop in enumerate(data.keys):
        for pub in self._output_publishers[prop]:
            pub.publish(data.values[idx])
        #output_manager_logger.debug('OUTPUT prop {0}:{1}'.format
        ... (prop, data.values[idx]))
```

The setup_publish_topics is to set the publish topics. In each project, there are several system nodes, each of them has a topic accordingly. This function is designed to initialize these topics.

```
def setup_publish_topics(self, prop_to_ros_info):
    for prop, prop_info in prop_to_ros_info.iteritems():
        if not prop in self._output_publishers.keys():
            self._output_publishers[prop] = []
        if isinstance(prop_info, list):
            for prop_info_element in prop_info:
```

```
                self._output_publishers[prop].append(rospy.Publisher
                ... (prop_info[0]['node_subscribe_topic'],
                ... std_msgs.msg.Bool,
                ... queue_size=10))
        else:
            self._output_publishers[prop].append(rospy.Publisher
            ...(prop_info['node_subscribe_topic'], std_msgs.msg.Bool,
            ... queue_size=10))
```

Any modification related to command message can be realized if the two functions above are changed properly.

BIBLIOGRAPHY

[1] http://http://wiki.ros.org/amcl {ROS package AMCL}, git: https://github.com/ros-planning/navigation.

[2] http://wiki.ros.org/move_base {ROS package move_base}, git: https://github.com/ros-planning/navigation.

[3] Rüdiger Ehlers and Vasumathi Raman. Slugs: Extensible GR(1) synthesis. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 333–339, 2016.

[4] C. Finucane, Gangyuan Jing, and H. Kress-Gazit. Ltlmop: Experimenting with language, temporal logic and robot control, git: https://github.com/verifiablerobotics/ltlmop/. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1988–1993, Oct 2010.

[5] Dieter Fox, Sebastian Thrun, Wolfram Burgard, and Frank Dellaert. Particle filters for mobile robot localization. 2001.

[6] Michael Huth and Mark Ryan. Logic in computer science: Modelling and reasoning about systems, 1999.

[7] Hadas Kress-Gazit, Georgios E. Fainekos, and George J. Pappas. Translating structured english to robot controllers. *Advanced Robotics*, 22(12):1343–1359, 2008.

[8] Shahar Maoz and Jan Oliver Ringert. Gr (1) synthesis for ltl specification patterns. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 96–106. ACM, 2015.

[9] Madhavan Mukund. Linear-time temporal logic and büchi automata. 1997.

[10] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[11] Vasumathi Raman, Cameron Finucane, and Hadas Kress-Gazit. Temporal logic robot mission planning for slow and fast actions. In *2012 IEEE/RSJ In-*

ternational Conference on Intelligent Robots and Systems, pages 251–256. IEEE, 2012.

[12] Vasumathi Raman, Nir Piterman, Cameron Finucane, and Hadas Kress-Gazit. Timing semantics for abstraction and execution of synthesized high-level robot control. *IEEE Trans. Robotics*, 31(3):591–604, 2015.

[13] Kai Weng Wong and Hadas Kress-Gazit. From high-level task specification to robot operating system (ros) implementation. In *2017 First IEEE International Conference on Robotic Computing (IRC)*, April 2017.

[14] Tichakorn Wongpiromsarn, Ufuk Topcu, Necmiye Ozay, Huan Xu, and Richard M. Murray. Tulip: A software toolbox for receding horizon temporal logic planning. In *Proceedings of the 14th International Conference on Hybrid Systems: Computation and Control*, HSCC '11, pages 313–314, New York, NY, USA, 2011. ACM.